“Lean and Efficient Software: Whole-Program Optimization of Executables”

Project Summary Report #4
(Report Period: 4/1/2014 to 6/30/2015)

Date of Publication: June 30, 2015

Sponsored by Office of Naval Research (ONR)

Contract No. N00014-14-C-0037
Effective Date of Contract: 06/30/2014

Technical Monitor:	Sukarno Mertoguno (Code: 311)
Contracting Officer:	Casey Ross

Submitted by:

GRAMMATECH

Principal Investigator: Thomas Johnson
531 Esty Street
Ithaca, NY 14850-4201
(607) 273-7340 x. 134
tjohnson@grammatech.com

Financial Data Contact:	**Administrative Contact:**
Krisztina Nagy	Derek Burrows
T: (607) 273-7340 x.117	T: (607) 273-7340 x.113
F: (607) 273-8752	F: (607) 273-8752
knagy@grammatech.com	dburrows@grammatech.com

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **30 JUN 2015**	2. REPORT TYPE	3. DATES COVERED **00-00-2015 to 00-00-2015**
4. TITLE AND SUBTITLE **Lean and Efficient Software:Whole-Program Optimization of Executables**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **GRAMMATECH,,531 Esty Street,,Ithaca,,NY, 14850**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**9**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

1 Financial Summary

Contract Effective Date	06/30/2014
Contract End Date	06/30/2016
Reporting Period	4/1/2014 – 06/30/2015
Total Contract Amount	$602,165
Incurred Costs this Period	$31,691
Incurred Costs to Date	$320,213
Est. Cost to Completion	$281,952

2 Project Overview

Background:
Current requirements for critical and embedded infrastructures call for significant increases in both the performance and the energy efficiency of computer systems. Needed performance increases cannot be expected to come from Moore's Law, as the speed of a single processor core reached a practical limit at ~4GHz; recent performance advances in microprocessors have come from increasing the number of cores on a single chip. However, to take advantage of multiple cores, software must be highly parallelizable, which is rarely the case. Thus, hardware improvements alone will not provide the desired performance improvements and it is imperative to address software efficiency as well.

Existing software-engineering practices target primarily the productivity of software developers rather than the efficiency of the resulting software. As a result, modern software is rarely written entirely from scratch—rather it is assembled from a number of third-party or "home-grown" components and libraries. These components and libraries are developed to be generic to facilitate reuse by many different clients. Many components and libraries, themselves, integrate additional lower-level components and libraries. Many levels of library interfaces—where some libraries are dynamically linked and some are provided in binary form only—significantly limit opportunities for whole-program compiler optimization. As a result, modern software ends up bloated and inefficient. Code bloat slows application loading, reduces available memory, and makes software less robust and more vulnerable. At the same time, modular architecture, dynamic loading, and the absence of source code for commercial third-party components make it hopeless to expect existing tools (compilers and linkers) to excel at optimizing software at build time.

The opportunity:
Our objective in this project is to substantially improve the performance, size, and robustness of binary executables by using static and dynamic binary program analysis techniques to perform whole-program optimization directly on compiled programs: specializing library subroutines, removing redundant argument checking and interface layers, eliminating dead code, and improving computational efficiency. In particular, we will apply specialization and partial evaluation technology, integrating the new technology with the techniques developed during the previous contract effort. We expect the optimizations to be applied at or

immediately prior to deployment of software, giving our tool an opportunity to tailor the optimized software to its target platform. Today, machine-code analysis and binary-rewriting techniques have reached a sufficient maturity level to make whole-program, machine-code optimization feasible. Thus, we believe there is now a great opportunity to design tools that will revolutionize the software development industry.

Work items:

We expect to develop algorithms and heuristics to accomplish the goals stated above. We will embed our work in a prototype tool that will serve as our experimental and testing platform. Because “Lean and Efficient Software: Whole-Program Optimization of Executables” is a rather long title, we will refer to the project as *Layer Collapsing* and the prototype tool as ***Laci*** (for **LA**yer **C**ollapsing **I**nfrastructure).

The specific work items for the base contract period are listed below:

1. **Investigate specialization opportunities.** The contractor will design and implement limit studies that will help focus the search for fruitful applications of partial evaluation and set goals for attainable improvements.
2. **Transfer UW technology.** The contractor will transfer program-specialization or partial-evaluation technology from the University of Wisconsin and integrate it into the contractor’s tool chain.
3. **Improve and extend UW technology.** The contractor will improve the robustness and scalability of the transferred technology, and complete partially implemented components and functionality.
4. **Improve and extend IR construction and rewriting.** The contractor will improve intermediate-representation construction and rewriting infrastructure as needed to demonstrate functionality on the primary test subjects.
5. **Develop and maintain test infrastructure.** The contractor will create an extensive suite of test applications, and will maintain and extend it as necessary. The contractor will also implement validation and measurement functionality that will enable tracking the robustness and benefits of program transformations.
6. **Investigate security implications.** As time permits, the contractor will study the effect of different instruction-generation mechanisms, such as peephole superoptimization, on security. As time permits, the contractor will also study whether polyvariant specialization enables (i) the creation of finer security-relevant models of program behavior and (ii) more accurate or efficient enforcement of security policies. If earlier tasks that are essential in completing a functional prototype require more effort, we propose to shift this task to the option period, with the possible adjustments of lower effort on either or both of the first two option-period tasks.

7. **Produce deliverables and attend required meetings.** The contractor will produce technical documentation in the form of reports and a working software prototype. The contractor will attend meetings requested by the program monitor.

3 Accomplishments during the reporting period

In the previous quarter we had begun transitioning UW's partial evaluation technology for use in LACI. This quarter we focused on exploring different options for how to apply the partial evaluator on a sizeable program. We examined both applying it at the whole-program level and trying to identify useful subcomponents on which to focus the partial evaluator.

A key challenge here is in constructing a partitioning of the input program into instructions that can be statically evaluated and those that require dynamic input. At the whole program level, it is difficult to implement this construction effectively due to the complexity of tracking dynamic data flow. We currently are achieving only a 3% division for the portion of the code marked static.

Operating on subcomponents seems a promising alternative, as dynamic data flow can be locally determined more precisely. We've made an initial stab at carving out subcomponents to operate on independently. This initial approach hasn't yielded much improvement. However, we're aware of a key flaw in the approach and plan to investigate a fix for this flaw in the next quarter.

A related problem that we've encountered has to do with the fact that the partial evaluator relies on analysis of control dependences in the original program. Our observation is that control dependence analysis is too conservative, resulting in a division between static and dynamic instructions being overly pessimistic (labelling instructions that can be evaluated statically as if they were dynamic). This is particularly problematic for functions that have early exit paths (e.g. for error checking). In the next quarter, we plan to investigate what options exist for relaxing the reliance on control dependences.

The following sections provide details on these accomplishments.

3.1 Application of Partial Evaluation at the Whole-Program Level

The partial evaluator involves three high-level steps. This first is to partition the instructions in the program into two sets: static and dynamic. The second step operates on the static instructions, performing the evaluation of their operations on a partial state of the program. The third step is to generate new code in which the static instructions are removed and replaced with sufficient code to feed in the correct state to the remaining dynamic instructions when they depend on values computed by the static instructions.

The effectiveness of the partial evaluator for a particular application will therefore be dependent on how large the static half of the partitioning is. Constructing the perfect partitioning is generally not possible due to imprecision in our ability to track all possible data flows (dependence analysis). Thus the partial evaluator errs on the conservative side, labelling some instructions as dynamic even though they could be statically evaluated.

A second challenge that affects the quality of the partitioning is how well we can identify sources of dynamic inputs. Again, to be conservative, we rely on an over-approximation of the actual sources of dynamic data in a program. For example, if a program calls a function that is external to the application, and the analysis engine has no information on the behavior of that function, we conservatively assume that the function's return value (and objects passed by reference to the function) may contain dynamically provided data.

Taken together, these two issue appear to compound at the whole-program level. We applied the partitioning to couple mid-sized applications and found that the static partitioning covered only 3% of the instructions. This is likely too small a margin to gain much optimization benefit.

3.2 Application of Partial Evaluation on Subcomponents

One of the opportunities we're hoping LACI can leverage is the interface between client and library code. Assuming developers often don't need the full generality present in a given library, it's likely that a lot of the library code can be trimmed down with partial evaluation.

With this in mind, we began investigating the possibility of applying the partial evaluator on subtrees of the call graph. In particular, we looked at function calls in which one or more of the parameters passed to the function is a constant value. If we can generate a customized version of the function (and its callees) based on those fixed parameters, then we can eliminate excess code and computation for that specific calling context.

Our first approach to this was to constrain the partial evaluator's partitioning implementation to just the functions that are transitively called from a given call site. Unfortunately, we hit a snag due to how the partitioning is implemented. The partitioning leverages CodeSurfer's dependence analysis, which computes the data and control relationships between different instructions in the program. Basically, the partitioning is constructed by performing a forward slice from the dynamic seeds of the program. Instructions covered by the slice are deemed dynamic. Those not covered by the slice are static.

Operating at the subcomponent level, we start the slice at the non-constant parameters to a specific function call. (We must also include other possible sources of dynamic input within the subcomponent as well.) The instructions not covered by the slice can be deemed static with respect to the specific calling context that we're interested in.

Where we ran into trouble, however, is that the default behavior of CodeSurfer's slicing operation is to continue following dependence relationships beyond the boundaries of the subcomponent. If the slice reaches a second call site to the same function with a different configuration of constant/non-constant parameters, then the slice will be muddied by this second call-site. As a result, we will again have an overly conservative partitioning of the code.

CodeSurfer has a couple other slicing modes that we experimented with, however, none of them quite have the right behavior needed in our situation. Instead, it seems the right approach is implement a customized slice using CodeSurfer's API that properly constrains the slice to the given calling context. We're planning to attempt this in the next quarter.

3.3 The Problem of Control Dependences

The partial evaluator leverages dependence analysis to construct the partitioning between static and dynamic instructions. This relies on two notions of dependence:

- A data dependence indicates that a value computed at one instruction is used at another instruction.
- A control dependence indicates that a control-flow operation performed at one instruction affects whether or not another instruction is executed.

Obviously, data dependences indicate direct flow of dynamic information through the system from one computation to the next. Control dependences represent more subtle flow of information. Consider the following function:

```
int foo(int input_var)
{
  int x;
  int y;
  if (input_var > 10) {
    x = 1;
  }
  else {
    x = 2;
  }
  y = x;
  return y;
}
```

In this example, the assignment to y has data dependences on both of the two assignments to x. However, neither of the assignments to x has any data dependences on any other instructions - their computation has no inputs. Thus, considering data dependence alone, the assignment to y does not appear to be dependent on dynamically provided input. Yet clearly its value does depend on the value of input_var. This is where control dependence comes in. Both of the assignments to x are control-dependent on the conditional of the if statement. Relying on both data and control dependences, we can correctly detect that the assignment to y should, in fact, be classified as a dynamic instruction.

However, leveraging control dependence results in a conservative over approximation of the set of instructions that should be considered dynamic. In the above example, both assignments to x would also be labelled dynamic, and thus the partial evaluator would ignore them. In this case, this wouldn't be too big of a problem. A more problematic case would be the following kind of idiom:

```
void bar(int input_var)
{
  if (!is_valid_input(input_var) {
    report_error();
    return;
  }

  /* ... rest of the function ... */
}
```

Here we have an error check at the beginning of the function to exit early if invalid data is provided. The "rest of the function" portion of the function is entirely control-dependent on this error check. Thus the slice performed by the partial evaluator will label this portion of the function entirely dynamic and miss any opportunities for optimization.

Other work in the literature has explored relaxing the requirement to rely on control dependence. A key problem that occurs when doing so is creating situations in which the partial evaluator fails to terminate (or provides an incorrect result). One approach to solving this is to use a termination analysis to attempt to determine if the partial evaluator will terminate in a given situation. If so, then the control-dependence can be relaxed. Another possibility is to pattern-match for specific situations (like the above) and relax the reliance on control dependence in such situations. We plan to explore these options in the next quarter.

4 Goals for the next reporting period

In the next reporting period we expect to complete the following:

- Investigate techniques for constraining the partial evaluator's slicing step in order to construct a more precise partitioning when operating on a subcomponent of a program.
- Investigate techniques for relaxing the reliance on control dependences when constructing the static/dynamic partitioning for the partial evaluator.

5 Milestones

Interim results on multi-month tasks will be reported in the quarterly progress reports.

Milestone	Planned Start date	Planned Delivery/ Completion Date	Actual Delivery/ Completion Date
Kickoff Mtg		9/4/2014	9/4/2014
Transition Specialization Slicing	7/2014	12/2014	12/2014
Robustness & Reliability of IR & Rewriting	7/2014	12/2014	12/2014 – statically linked exes
First Quarterly Report		9/30/2014	11/21/14
Transition Partial Evaluation and Instruction Synthesis	12/2014	5/2015	In progress
Second Quarterly Report		12/30/2014	2/19/2015
Third Quarterly Report		3/30/2015	5/11/2015
Fourth Quarterly Report		6/30/2015	7/3/2015
Fifth Quarterly Report		9/30/2015	
Sixth Quarterly Report		12/30/2015	
Seventh Quarterly Report		3/30/2016	
Evaluation	4/2016	6/2016	
Final Report		6/30/2016	

6 Issues requiring Government attention

None.